TAMPEREEN
AMMATTIKORKEAKOULU

REST API CLIENT -OHJELMAKIRJASTO POTILASTIETOJÄRJESTELMÄÄN

Roope Rantanen

Opinnäytetyö
Toukokuu 2018
Tietojenkäsittely
Ohjelmistotuotanto

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittely
Ohjelmistotuotanto

RANTANEN, ROOPE:
REST API client-ohjelmakirjasto potilastietojärjestelmään

Opinnäytetyö 43 sivua
Huhtikuu 2018

Opinnäytetyön toimeksiantajana oli Vitec Softwaren Tampereen yksikkö, joka kehittää Acute-potilastietojärjestelmää. Opinnäytetyön tarkoituksena oli löytää keino paketoida Acuten REST-rajapinnan toiminnallisuudet ohjelmakirjastoksi ulkopuolisten kehittäjien käyttöön. Opinnäytetyön tarkoituksena oli tämän lisäksi suunnitella ohjelmakirjaston ylläpito- ja julkaisuprosessi.

Opinnäytetyön tavoitteena oli rakentaa työkalu, joka helpottaa mahdollisimman paljon kolmansien osapuolien kehitystyötä Acuten rajapinnan kanssa. Projektin tuotoksella oli tarkoitus alentaa kynnystä kolmansille osapuolille palveluiden tuottamiseen. Koska uusien ohjelmistokokonaisuuksien ylläpito vaatii usein lisäresursseja, tutkimuksessa painotettiin työkalun automatisointia ylläpidon ja julkaisun kannalta.

Tutkimuksen alussa toimeksiantajan kanssa käytiin määrittelypalavereita, joilla kartoitettiin tarvittavan työkalun vaatimukset. Näistä vaatimuksista syntyi lista, jonka avulla aloitettiin kehityksen suunnittelu. Suunnittelu lähti käyntiin selvittämällä olemassaolevia ratkaisuja REST-rajapintojen käyttämiseen ja miettimällä miten näitä voitaisiin hyödyntää kehitysprosessissa. Suunnitelman valmistuttua se esiteltiin toimeksiantajalle, jonka hyväksymisen jälkeen kehitystyö aloitettiin. Kehitystyön aikana kiinnitettiin huomiota määrittelyissä toivottuihin vaatimuksiin, kuten ylläpidon automatisointiin. Viimeisenä vaiheena projektissa oli suunnitelman kirjoittaminen ohjelmakirjaston julkaisuprosessista.

Opinnäytetyön tuotoksena syntyi automatisoitu prosessi, joka mahdollisti lähdekoodista generoituvan ohjelmakirjaston luonnin. Prosessi perustuu OpenAPI-spesifikaation teknologiaan, josta erillisellä työkalulla voidaan luoda ohjelmakirjastoja. Tutkimuksesta syntyi myös suunnitelma ohjelmakirjaston automatisoidusta julkaisuprosessista. Tutkimuksen tuotos vastasi toimeksiantajan asettamia vaatimuksia työkalun helposta ylläpidosta sekä julkaisusta. Toimeksiantajalle lisäarvona oli myös mahdollisuus käyttää opinäytetyön tuloksia sisäisissä koulutuksissa.

Asiasanat: rest, ohjelmakirjasto, rajapinta, automatisointi

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in Business Information Systems
Software Development

RANTANEN, ROOPE:
Implementing REST API-Client Library
For Patient Infromation System

Bachelor's thesis 43 pages
April 2018

This thesis was commissioned by Vitec Software's unit located in Tampere. The unit in Tampere provides the patient information system Acute for a vast number of different healthcare organizations. The purpose of this thesis was to find a way to wrap Acute's REST API to a software library, which would further attract third party organisations to develop services around Acute's ecosystem. The emphasis on the requirements of the library was in ease of library's upkeep and deployment process. In addition to the librarys development project, the thesis also researches the process of deploying and publishing the library to the public. Each process was hoped to be as automated as it could be. The methods for the project varied from specification meetings to the use of automated tools for consuming REST API's

As a product of this thesis an automated process of generating a software library from Acute's REST API was created. The process utilises OpenAPI-specification technology, which enables the automated generation from source code with minimal effort in upkeeping. The research also produced a plan for automated process for deploying and publishing the library. In addition to achieving the commissioned goal, additional value is achived by using this thesis as a learning material in internal training.

Key words: rest, software, library, api, automation

SISÄLLYS

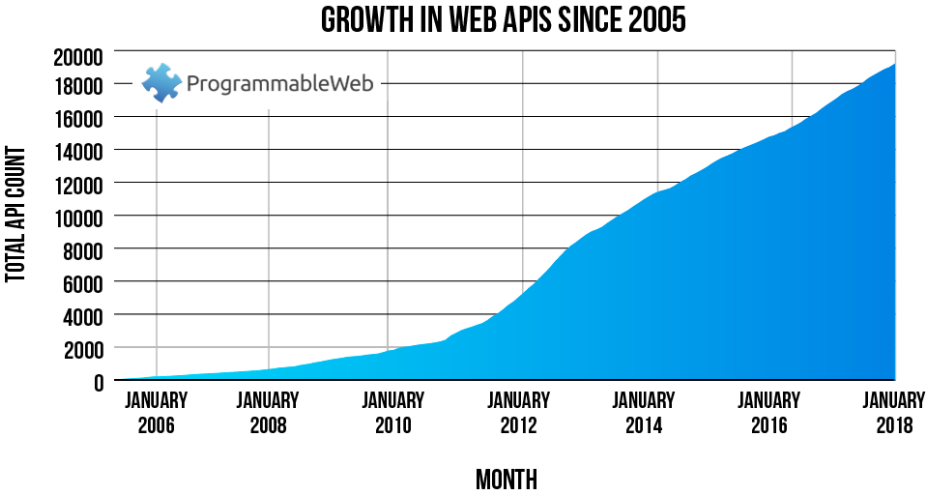
LYHENTEET JA TERMIT

API	Application programming interface. Ohjelmointirajapinta.
Git	Versionhallintaohjelmisto
HTTP	Hypertext Transfer Protocol. Tiedonsiirtoprotokolla.
JSON	JavaScript Object Notation. Avoimen standardin tiedosto-muoto tiedonvälitykseen
OAS	OpenAPI Specification. Kuvauskieli ohjelmistorajapintojen kuvaamiseen koneluettavassa muodossa.
REST	Representational State Transfer. HTTP-protokollaan perus-tuva arkkitehtuurimalli rajapintojen toteuttamiseen.
SOAP	Simple Object Access Protocol. Tiedonsiirtoprotokolla.
URI	Uniform Resource Identifier. Merkkijono joka osoittaa tieto-resurssin osoitteen.
XML	Extensible Markup Language. Rakenteellinen kuvauskieli tietomassojen jäsentämiseen.

1 JOHDANTO

1.1 Opinnäytetyön tausta, tavoite ja rajaukset

Ohjelmistotuotannossa yleinen ongelma on kehityksen kohtaamat kasvukivut. Kun kehitettävän sovelluksen toiminnallisuudet lisääntyvät, kasvaa riski muutosten aiheuttamista virhetilanteista merkittävästi. Sovelluksien monoliittisuutta pyritään estämään määrittelemällä sovelluksille komponenttiarkkitehtuureja ja rajapintoja. Rajapintojen käyttö ohjelmistojen välillä on ollut laajalti kasvava ilmiö jo vuosien ajan. Lisäämällä rajapintoja eri kokonaisuuksien välille, saadaan ohjelmistosta helpommin ylläpidettävä, kun osat toimivat itsenäisinä kokonaisuuksina (Alfame n.d. 5). Nykyään rajapintojen maailmaa hallitsee REST-arkkitehtuurimalli, joka on nousunsa myötä helpottanut rajapintojen implementoimista maailmalla. Rajapintojen hallitseva tietolähde ProgrammableWeb-sivusto listasi vuodesta 2005 vuoteen 2018 tietokantaansa lisättyjen rajapintojen määrän (Kuva 1). Kuvasta voidaan nähdä, kuinka merkittävä kasvu rajapintojen kehityksessä on tapahtunut vuosien saatossa.



KUVA 1. ProgrammableWeb sivuston tietokantaan lisätyt rajapinnat vuosina 2005-2018

Rajapintojen maailmassa kasvava trendinä on tarjota myös työkaluja rajapinnan helpompaan käyttöön. Yritykset kuten Paypal ja Amazon tarjoavat kehittäjille ohjelmakirjastoja, jotka helpottavat rajapintojen käyttöönottoa omissa sovelluksissa. Näitä ohjelmakirjastoja tarjoamalla pyritään madaltamaan kynnystä palveluntarjoajille sekä saamaan uusia

palveluita sovelluksen ympärille. Ohjelmakirjastot vähentävät palveluntarjoajien kehitystyötä sisältämällä rajapinnan toiminnot ja kommunikaatiologiikan helposti käytettävässä muodossa.

Ohjelmakirjaston tarjoamien hyötyjen takia on todettu tarpeelliseksi paketoida myös toimeksiantajan tarjoama REST-rajapinta C#-kirjastoksi. Tavoitteena kirjastolle on pitää kirjaston ylläpito mahdollisimman nopeana ja suoraviivaisena. Opinnäytetyön pääasiallinen tavoite on kehittää prosessi ohjelmakirjaston luomiseen ja julkaisuun. Kirjaston tarkoitus on helpottaa mahdollisimman paljon kolmansien osapuolien kehitystyötä Acuten rajapinnan kanssa, jonka kanssa työskentely saatetaan kokea ajoittain vaikeaselkoiseksi muuttuvan dokumentaation vuoksi. Syntyvällä tuotoksella pyritään kolmansille osapuolille tarjoamaan ohjelmakirjasto, joka nopeuttaa palveluiden kehitysaikoja ja vähentää rajapinnan muutoksista aiheutuvia virhetilanteita.

Toimeksiantajan motivaatio ulkopuolisten kehittäjien houkuttelulle löytyy laajentumisesta. Kun Acuten ympärille saadaan palveluntarjoajia kehittämään erilaisia palveluita, se saa enemmän jalansijaa jatkuvasti kehittyvien potilastietojärjestelmien markkinoilla. Projektin sisältö on kolmivaiheinen. Projektiin kuuluu tutkimus keinoista saavuttaa haluttu työkalu, työvaiheiden suunnittelu ja konkreettinen toteutus. Suunnittelussa ja toteutuksessa painotetaan kirjaston geneerisyyttä, jossa kirjasto päivittyy mahdollisimman automaattisesti rajapinnan päivittyessä. Konkreettisessa toteutuksessa pyritään todentamaan työvälineen käyttötapaukset ja toimiminen.

1.2 Toimeksiantaja

Vitec Software on laaja pohjoismainen organisaatio, johon kuuluu lukuisia eri aloille erikoistuneita yksiköitä. Tampereen yksikkö Acuvitec Oy kehittää Acute nimeä kantavaa potilastietojärjestelmää. Acuvitecin työntekijöiden lukumäärä on noin 50 henkilöä ja toimitilat sijaitsevat Tampereen keskustassa. Acuvitecin päätarkoitus on kehittää ja toimittaa potilastietojärjestelmää nimeltään Acute, ja tarjoaa samalla rajapintoja mahdollistamaan kolmansien osapuolien tuotteiden kehityksen Acuten rinnalle.

2 ACUTE REST RAJAPINTA

2.1 Tausta

Vitec Softwaren kehittämä Acute-potilastietojärjestelmä on yksi Suomen johtavista web -pohjaisista potilastietojärjestelmistä. Acute-tietojärjestelmän tuotemallissa suositaan laajenevissa määrin modulaarista ohjelmistokehitystä, jossa Acuten rinnalle voidaan kehittää itsenäisiä toimintokokonaisuuksia. Tarkoituksena on tuoda asiakkaille mahdollisuus hankkia Acuten kanssa toimivia komponentteja päätuotteen rinnalle, ja näin laajentamaan Acuten toiminnallisuutta (Hautamäki 2017). Tätä tuotemallia varten Acuten ympärille on kehitetty kolmannen osapuolen kehittäjiä varten Acusfääri-ekosysteemi, joka koostuu Acuten ympärille rakennetuista kolmannen osapuolen palveluista. Nämä palvelut hyödyntävät aktiivisesti erilaisia liittymämalleja sekä Acuten omaa REST-rajapintaa. Asiakas voi esimerkiksi laskutusta tai nettiajanvarausta varten hankkia oman sovellusratkaisun toiselta ohjelmistotoimittajalta. Asiakas voi myös vaihtaa potilastietojärjestelmänsä Acuteen vaihtamatta kuitenkaan kyseisiä komponentteja. Rajapintojen avulla nämä toiminnallisuudet saadaan toimimaan saumattomasti yhdessä Acute-potilastietojärjestelmän kanssa.

Acuten REST-rajapinta käsittää laajan kirjon erilaisia terveydenhuollon päivittäisessä käytössä yleisiä kyselyitä ja toimintoja. Nämä ovat esimerkiksi potilastietojen ja organisaatioasiakkaiden työntekijätietojen haku, toimitilojen varauksien muodostaminen, laskujen teko sekä potilasajanvaraukset. Rajapinnan avulla voidaan esimerkiksi hakea koko organisaatiohierarkian tiedot (kuva 2).

```
{
    "employer": "/reservers/1000267/employers/100008",
    "name": "Pellon Takamaan tehdas, Henkilöstöhallinto (Emo)",
    "parent": "/reservers/1000267/employers/100407",
    "hasChildren": true,
    "defOrg": false,
    "employerStatus": 4,
    "employerStatusDescr": "Tuotannossa, laskutus + kertomus"
},
{
    "employer": "/reservers/1000267/employers/100007",
    "name": "Pellon Takamaan tehdas, Henkilöstöhallinto, Palkanlaskenta",
    "parent": "/reservers/1000267/employers/100008",
    "hasChildren": false,
    "defOrg": false,
    "employerStatus": 4,
    "employerStatusDescr": "Tuotannossa, laskutus + kertomus"
},
{
    "employer": "/reservers/1000267/employers/100009",
    "name": "Pellon Takamaan tehdas, Henkilöstöhallinto, Työhönotto",
    "parent": "/reservers/1000267/employers/100008",
    "hasChildren": false,
    "defOrg": false,
    "employerStatus": 4,
    "employerStatusDescr": "Tuotannossa, laskutus + kertomus"
},
```


KUVA 2. Lyhennetty esimerkki palautuneesta organisaatiohierarkia-tulosjoukosta

2.2 Käyttömahdollisuudet

Alun perin Acuten REST-rajapinta rakennettiin pääsääntöisesti verkkoajanvarausta varten. Acuten terveydenhuollon asiakkaat pystyivät halutessaan hankkimaan ajanvarausratkaisun toiselta palveluntarjoajalta. Asiakkaat pystyivät myös siirtymään Acuten potilastietojärjestelmään vaihtamatta olemassa olevaa ajanvarausratkaisuaan integroimalla tämän ratkaisun toimimaan yhdessä Acuten kanssa. Rajapinta on ajan myötä yhteistyössä Acuten asiakkaiden kanssa kuitenkin laajentunut jatkuvasti yhä kattavammaksi kokonaisuudeksi, eikä laajentumiselle ole näkyvissä hidastumisen merkkejä. Terveydenhuollon ja teknologian kehittyessä rajapintaan muodostuu jatkuvasti yhä uusia tarpeita kehitykselle.

Terveydenhuolto on vuonna 2018 siirtymässä yhä laajenevissa määrin tarjoamaan asiakkailleen digitaalisia palveluja tavanomaisen terveydenhuollon rinnalle. Vuonna 2016 Kela rinnasti terveydenhuollon etäpalvelut perinteisiin vastaanottokäynteihin, jolloin potilas saa niistä täyden Kela-korvauksen (Koho, S. 2017). Uusiin digitaalisiin palveluihin lukeutuvat esimerkiksi videovälitteiset lääkärinvastaanotot, joissa terveydenhuollon ammattilainen kykenee hoitamaan asiakkaita ilman näiden fyysistä läsnäoloa. Palveluihin

kuuluvat myös terveys- ja hyvinvointipalveluiden mobiilisovellukset, joilla asiakas saa yhteyden terveydenhuollon ammattilaiseen. Mobiilisovellukset mahdollistavat asiakkaalle terveydenhuollon palveluja ilman terveyskeskukseen matkaamista. Nämä julkisen ja yksityisen terveydenhuollon uudet tarpeet Acute pyrkii mahdollistamaan REST-rajapintansa avulla.

3 REST-ARKKITEHTUURI

3.1 RESTful periaate

REST (Representational State Transfer) on HTTP-protokollaan perustuva arkkitehtuurimalli ohjelmointirajapintojen toteuttamiseen. REST-arkkitehtuurimallin kehitti Roy Fielding vuonna 2000 osana väitöskirjaansa. Fieldingin tavoitteena oli luoda standardi kahden palvelimen väliselle kommunikaatiolle ja tiedonsiirrolle niiden sijainnista riippumatta. REST arkkitehtuurimallissa data ja toiminnallisuudet ajatellaan resursseina ja näitä aksessoidaan linkkimuodossa olevien URI:en (Uniform Resource Identifier) kautta (Oracle 2013). REST kehittyi vaihtoehdoksi SOAP-teknologialle (Simple Object Access Protocol), joka perustuu XML-kieleen. Teknologiana SOAP hallitsi laajalti rajapintojen maailmaa pitkälle 2000-luvun alkuun. Nykyään REST on syrjäyttänyt SOAPin hallitsevana uusien rajapintojen teknologiana (Mason 2011).

RESTin päätarkoitus on mahdollistaa Internetin välityksellä toimivien sovellusten yhteentoimivuus noudattamalla ennalta määrättyä standardia. Tämä mahdollistaa sovellusten sisäisen kehityksen ilman yhteentoimivuuden kärsimistä. REST-arkkitehtuurimalli rakentuu ennalta määrättyjen ominaisuuksien, periaatteiden ja rajoituksien kokonaisuudesta, jotka mahdollistavat RESTin hyvän suorituskyvyn, luotettavuuden sekä laajennettavuuden. Jotta rajapinnan voidaan todeta noudattavan RESTful-periaatetta, tulee sen noudattaa tämän kokonaisuuden osa-alueita.

3.2 Arkkitehtuurimallin rajoitteet

REST arkkitehtuurin rajoitukset mahdollistavat arkkitehtuurimallin ominaiset piirteet. Näihin piirteisiin kuuluvat hyvä suorituskyky, skaalautuvuus ja luotettavuus. REST-arkkitehtuurimallin luoja Roy Fielding määritteli vuonna 2000 kirjoittamassaan väitöskirjassaan nämä rajoitteet, joiden päälle koko RESTin malli rakentuu.

3.2.2 Asiakas-palvelin

REST-arkkitehtuuriin ensimmäisien joukossa lisätty rajoite on asiakkaan (client) ja palvelimen (server) toimintojen eroittaminen toisistaan. Tämä perustuu ajatukseen, jossa jakamalla toiminnallisuudet toisistaan parannetaan käyttöliittymän siirrettävyyttä monelle eri alustalle ja parannetaan skaalautuvuutta pitämällä palvelinpuolen komponenttien rakenne yksinkertaisempana (Fielding 2000. 78). Asiakkaan tarkoitus on toimia kyselyiden syöttäjänä palvelimelle. Palvelimen tarkoitus on toteuttaa kyselyiden määrittämät operaatiot ja palauttaa vastaus toteutetusta operaatiosta takaisin asiakkaalle. Merkittävä etu tässä toimintatavassa on näiden kahden osapuolen riippumattomuus toisistaan sisäisen kehityksen kannalta. Palvelinta ja asiakasta voidaan kehittää vapaasti, kunhan rajapinta ei muutu. Asiakas ja palvelin voivatkin itsenäisesti käyttää mitä tahansa ohjelmointikieltä tai teknologiaa, kunhan molemmat noudattavat yhteistä rajapintamallia (Massé 2012. 3).

3.2.3 Tilattomuus

Tilattomuuden (Stateless) rajoite tarkoittaa REST-arkkitehtuurimallissa periaatetta, missä jokaisen kyselyn täytyy sisältää kaikki tarvittava tieto kutsun ymmärtämiseksi palvelimen päässä. Tämä tarkoittaa käytännössä sitä, että asiakasohjelma ei saa tilanteesta riippumatta perustaa toimintaansa palvelimen edeltävistä kutsuista tai istunnon aikana mahdollisesti tallentamaan dataan. Asiakasohjelman on aina säilytettävä kaikki tieto istunnon tilasta itsellään, ja syötettävä kaikki halutun operaation vaatimat tiedot kyselyä tehdessään. Tämä vaatimus mahdollistaa kuorman vähentämisen palvelimen puolella, sillä sen ei tarvitse varastoida dataa kyselyjen välillä. Palvelin ei myöskään joudu päättelemään osittaisen kyselyn perusteella haluttua operaatiota ja sen tulosta. Yhtenä tilattomuuden haittapuolena voidaan pitää tiedonsiirron kasvua, operaatioihin tarvittavien tietojen kulkiessa toistuvasti jokaisen kyselyn sisällä (Fielding 2000. 78-79.)

3.2.4 Välimuistitus

Helpotusta tilattomuuden tuoman tiedonsiirtomäärän vähentämiseen tuo arkkitehtuurimallin toinen rajoite välimuistitus. Toistuvien kyselyiden sijasta palvelimelle, voidaan REST-arkkitehtuurimallissa käyttää asiakasohjelman välimuistiin tallennettuja kyselyitä.

Mikäli kyselyn vastaukselle on merkitty tieto sen tallentamisesta välimuistiin, voi asiakasohjelma tämän jälkeen uutta samanlaista kyselyä tehdessään käyttää tätä välimuistiin tallennettua vastausta uudelleen. Näin saadaan vähennettyä tietoliikennettä palvelimen ja asiakkaan välillä. Tämä toiminnallisuus ehkäisee palvelimelle kohdistuvia toistuvia kyselyitä ja parantaa täten suorituskykyä. Rajoitteen merkittävin haittapuoli luotettavuuden suhteen tulee kuitenkin tilanteesta, jossa kyselyn vastauksena käytettävä, välimuistissa oleva data on ehtinyt muuttua palvelimen päässä (Fielding 2000. 80.)

3.2.5 Yhdenmukainen rajapinta

REST-arkkitehtuurimallin tärkeimpiä periaatteita on rajapintojen pitäminen yhtenäisenä. Tämä tarkoittaa lyhyesti muotoiltuna sitä, että kyselyt ja vastauksena palautuvan tulosjoukon resurssit noudattavat rakenteeltaan aina samaa kaavaa. Tähän periaatteeseen kuuluvat myös HTTP-protokollasta tuttujen metodien kuten GET-, PUT-, POST- ja DELETE käyttö, jotka määrävät minkäluontoisen operaation kyselyn on tarkoitus tehdä. REST-malliin kuuluu näiden metodien oikeanlainen käyttö, sillä esimerkiksi POST-metodilla voidaan toteuttaa lähes kaikki muiden metodien tekemät operaatiot. Metodien käyttötarkoitukset ovat karkeasti seuraavat:

- POST-metodia käytetään resurssin luomiseen palvelimella.
- GET-metodia käytetään resurssin hakemiseen palvelimelta.
- PUT-metodia käytetään resurssin päivittämiseen palvelimella.
- DELETE-metodia käytetään resurssin poistamiseen palvelimelta.

R. Fieldingin yhtenä pääajatuksena REST-mallissa on pitää URI samana, metodien määrätessä tehtävän operaation. Jos käyttäjä haluaa hakea asiakkaan tiedot, tulee hänen tehdä kysely HTTP-metodilla GET /clients/123, missä 123 on asiakkaan yksilöivä tunnusnumero. Jos käyttäjä taas haluaa poistaa kyseisen asiakkaan, tulee hänen tehdä kysely samalla URI:lla /clients/123, mutta käyttää DELETE-metodia (Fielding 2000. 81.)

Jotta rajapintaa voitaisiin pitää RESTful-arkkitehtuuria noudattavana, yhtenä tärkeimpänä asiana on rajapinnan jakaminen resursseihin, jossa rajapinnan jokainen palanen yksilöidään omalla URI:lla (Richardson & Ruby 2007. 303). Mikäli kyselyssä palautuvan resurssin sisältä löytyy viittaus toiseen resurssiin, esitetään se samanlaisessa yksilöivässä

muodossa kuin alkuperäinen kysely on tehty. Tällä tavoin käyttäjä pystyy tekemään yhdenmukaisesti rakentuvia kyselyitä sen sijaan että joutuisi olemaan tietoinen jokaisen erilaisen kyselyn spesifeistä muodoista (Fielding 2000. 81.)

Käyttötapausesimerkissä käyttäjä tekee kyselyn asiakkaan tiedoista käyttämällä kutsua GET /clients/123, jossa 123 on asiakkaan yksilöivä tunnusnumero. Palautuvassa tulosjoukossa yhtenä asiakkaan tiedoista kuvataan asiakkaan työnantaja, jonka resurssin yksilöivä tunnusnumero on 111. REST-periaatetta noudattamalla tämä työnantaja -resurssi tulee esittää muodossa "/employers/111", jolloin käyttäjän on helppo muodostaa uusi kutsu muodossa GET /employers/111 ja päästä käsiksi työnantaja -resurssin tietoihin.

3.2.6 Kerroksellinen järjestelmä

Kerroksellinen järjestelmä RESTin rajoituksena tarkoittaa sitä, että vaikka asiakasohjelma tekee kyselyitä tietylle palvelimelle, voi tämä palvelin olla vain yksi monista eri palvelinpuolen tasoista. Palvelimet voivat välittää kyselyä eteenpäin aina seuraavalle palvelinpuolen komponentille, kunnes kyselyn pyytämän operaation suorittava komponentti saavutetaan. Tämä mahdollistaa esimerkiksi kyselyiden reitityksen palomuurien kautta sekä ohjauksen ohjelmiston eri versioita tukeviin rajapintoihin. Kerroksellisen järjestelmän tuomiin hyötyihin voidaankin lukea skaalautuvuuden parannus, sillä kyselykuorman kasvaessa kuormaa kyetään jakamaan eri palvelimille. Myös ohjelmiston monimutkaisuus vähenee toiminnallisuuden jakautuessa omiin komponentteihinsa. Haittapuolena kerroksellisuudessa on suorituskyvyn lasku kyselyn matkatessa usean eri komponentin välillä vaatien näin enemmän prosessointivoimaa ja kasvattaen latenssia (Fielding 2000. 83.)

3.2.7 Koodin latauksen mahdollistaminen

REST-arkkitehtuurin viimeinen ja valinnainen rajoitus on mahdollisuus laajentaa asiakasohjelmaa lataamalla suoritettavaa koodia ja skriptejä suoraan palvelimelta. Tämä mahdollistaa käyttäjälle hyödyllisten ohjelmakoodien lataamisen palvelimelta suoraan, joutumatta itse implementoimaan näitä toimintoja asiakasohjelmaan. Tämän rajoituksen tarkoitus onkin yksinkertaistaa asiakasohjelmien rakennetta tarjoamalla valmiiksi toimivia

ominaisuuksia käyttäjälle. Koodin latauksen mahdollistamalla palvelinpuoli pystyy esimerkiksi ulkoistamaan täysin avaimien ja muiden kriittisten tietojen käsittelyn tarjoamalla näiden käsittelyyn tarkoitetun komponentin rajapinnan kautta asiakasohjelmalle (Fielding 2000. 84-85.)

4 API CLIENT POTILASTIETOJÄRJESTELMÄN RINNALLE

4.1 Tutkimusasetelma

Kun tutkimusta käytettävistä menetelmistä alettiin toteuttaa, kartoitettiin projektin kannalta oleellisimmat tutkimuskysymykset. Tämä kartoitus tapahtui haastattelemalla määrittelypalavereissa toimeksiantajaa ja yrityksen ammattilaisia. Nämä kysymykset listattiin seuraavalla tavalla.

- Miten rajapinta voidaan paketoida ohjelmistokirjastoksi?
- Miten kirjaston kehitys voidaan automatisoida vastaamaan päivittyvää rajapintaa?
- Miten kirjasto voidaan julkaista käyttäjille?

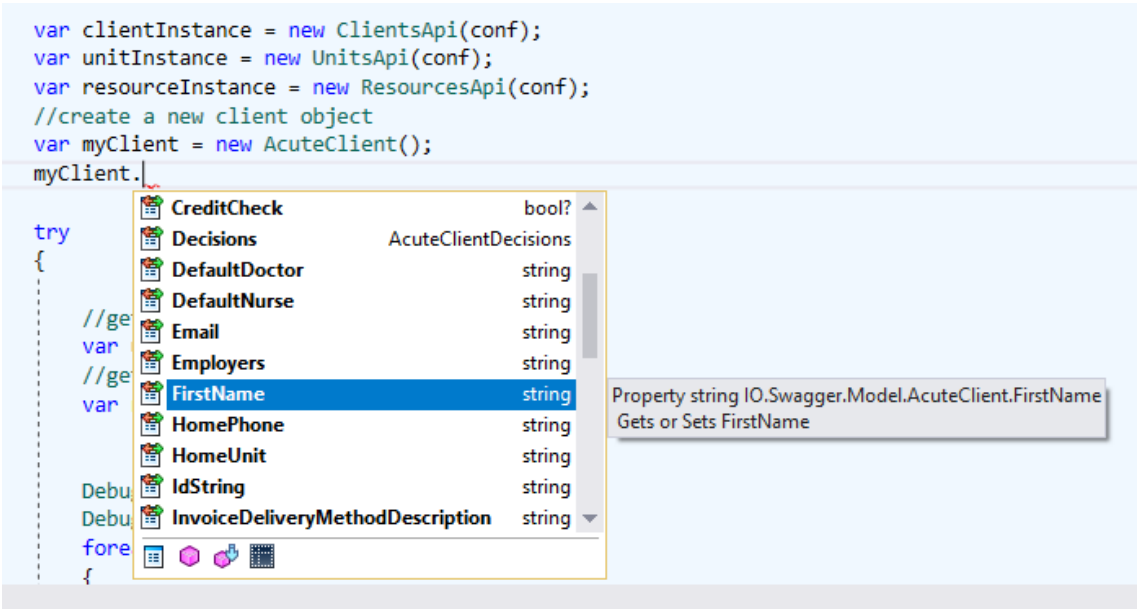
Jotta näihin kysymyksiin voidaan saada vastaus, on aiheellista avata ensin mitä rajapinnan toiminnallisuudet sisältävä ohjelmakirjasto tarkoittaa ja mikä on Acuten tämän hetkisen rajapinnan tilanne.

4.2 API client

Kuten aikaisemmin jo mainittiin, REST-malli perustuu HTTP-protokollaan. REST-kyselyt tehdään lähettämällä HTTP-kutsuja palvelimelle. Jotta näitä kutsuja voidaan tehdä, on kyselyn tekijän osattava rakentaa URI kohdistettavalle resurssille, ja täyttää kyselystä riippuen myös mahdollinen kyselyn rakenne (body), joka sisältää kyselyn tietosisällön. Tämän tietosisältö koostuu käytettävän REST-rajapinnan dokumentaation määrittelemistä atribuuteista. Kutsun muuttaessa palvelimella olevaa dataa, kyselyn rakenteen täyttää asiakasohjelma. Palvelimelta palautuvien kyselyvastausten rakenteen täyttää palvelinpuoli. Tämä prosessi tarkoittaa REST-rajapintaa hyödyntäville kehittäjille usein merkittävää määrää manuaalista työtä. Kun sovellukseen implementoidaan rajapintaa kutsuva logiikka, tulee kehittäjän toteuttaa rajapintakutsut dokumentaation määrittelemällä tavalla. Dokumentaation ja kutsujen vaatimien tekniikoiden sisäistäminen viekin usein merkittävän määrän resursseja kehitysprojektissa.

API client on REST-rajapinnasta muodostettu ohjelmakirjasto, joka sisältää rajapinnalle määritellyt resurssikyselyt sekä tiedon näiden resurssien sisältävistä tietosisällöistä. Täl-

läisen ohjelmakirjaston perimmäinen tarkoitus on helpottaa sovelluskehitystä vähentämällä kyselyiden manuaalista konfiguroimista teknisellä tasolla. Rajapinnan käyttö ohjelmakirjaston kautta nopeuttaa myös rajapintojen käyttöönottoa ohjelmistoprojektissa (Sarrel 2016). Kirjaston hyödyntäminen vähentää myös tarvetta rajapinnan dokumentaation sisäistämisestä, sillä kirjasto tarjoaa helpottavia ominaisuuksia dokumentaation noudattamiseen. Kun rajapintakyselyt paketoidaan ohjelmakirjastoksi, mahdollistuu esimerkiksi koodin täydennys (code completition) (kuva 3).



KUVA 3. Ohjelmistokirjaston koodin täydennys

Kehittäjä kirjoittaa resurssiluokan sovellukseensa, ja ohjelma näyttää automaattisesti listan kaikista resurssiluokan attribuuteista ja metodeista. API clientissä HTTP-protokollasta tutut GET-, POST-, PUT- ja DELETE- kyselyt resursseille on mahdollista toteuttaa metodin muodossa noudattaen ohjelmakirjaston kielen mukaista tyyliä, jolloin kutsujen teko rajapintaan ei ulkoisesti eroa normaaleista, käytettävän ohjelmointikielen syntaksin mukaisista metodikutsuista (kuva 4).

KUVA 4. Ohjelmakirjaston ehdottamat parametrit dokumentaation mukaisina

REST-rajapinnan käyttö kirjaston kautta tuo myös muita kehitystä helpottavia ominaisuuksia kehittäjille. Vaikka REST-rajapintojen käyttö on suoraviivaista, ohjelman ja rajapinnan väliseen interaktioon tarvittavan logiikan implementoiminen saattaa olla vaivalloista. Yhtenä ohjelmakirjaston hyötynä on yhteys- ja autentikoimislogiikan tarjoaminen kirjaston sisällä, jotta kirjastoa käyttävän kehittäjän ei tarvitse itse implementoida näitä komponentteja.

4.3 API clientin haasteet

API clientin haittapuolena on itse kirjaston tarjoajan kannalta se, että jokainen ohjelmointikieli vaatii oman kirjastonsa. Ohjelmakirjastoja käytetään ohjelmistojen modulaarisessa kehittämisessä, joka tarkoittaa käytännössä sen olevan moduuli eli kehitettävän ohjelman jatke joka laajentaa ohjelman toiminnallisuutta. Jotta kirjasto on ohjelman jatke, se on toteutettu samalla kielellä kuin ohjelma. Kun tavanomaiset REST-kutsut toteutuvat HTTP-kutsujen kautta, jolloin rajapinnan käyttäjä on toteuttanut näiden kutsujen operoinnin itse haluamallaan kielellä, ei rajapinnalle dedikoidun, kaikille kielille yhteensopivan kirjaston tarjoaminen ole mahdollista. Kun yleiseen käyttöön tarkoitettua ohjelmakirjastoa kehitetään REST-rajapinnalle, tarkoittaa tämä tarvetta kirjoittaa sama kirjasto kaikille niille kielille, millä rajapinnan toimintoja halutaan käyttää. Tämän haittapuolen poistamiseksi onkin alettu tuoda erilaisia ratkaisuja sovelluskehityksen piiriin, jotka mahdollistavat API clienttien generoimisen eri ohjelmointikielille suoraan rajapintojen lähdekoodista. Generoituvilla ohjelmakirjastoilla saavutetaan yhtenäinen toiminnallisuus ja kirjastojen ylläpitokuorma rajapinnan muuttuessa vähenee merkittävästi.

Kun tarkastellaan kehittäjien kannalta REST-rajapinnan paketointia ohjelmakirjastoksi ja sen tarjoamista kehittäjille, tulee huomioida mahdolliset kompastuskivet ja ratkaisut kirjaston tarjoamisen ongelmiin. Siinä missä kirjaston käyttö lyhentää kehitysaikoja merkittävästi, sen käyttö saattaa tuntua kehittäjästä rajoittavalta, mikäli sen jakelu on rajoittunut binäärimuodossa olevaan kirjastoon. Jos rajapinnan kanssa interaktio tapahtuu vain binäärimuodossa olevan kirjaston kautta, poistaa se käyttäjältä helposti saatavan tiedon sen sisältämistä toiminnoista, joka aiheuttaa tunnetta kontrollin puuttesta. Tässä tilanteessa myös mahdollisuus kirjaston debuggaukseen on suljettu pois. Kontrollin ja tiedon puute voidaan ratkaista sallimalla rajapinnan käyttö myös suorilla kutsuilla ilman kirjastoa, jolloin työmäärä kasvaa mutta kontrolli tehtyihin operaatioihin säilyy. Toisena ratkaisuna on tarjota kirjasto lähdekoodi-formaatissa. Tällöin kirjaston käyttäjä saa kirjaston koodin täysin omaan käyttöönsä, ja voi debugata kirjastoa ja halutessaan jopa muokata kirjastoa omaan kehitettävään ohjelmistoon sopivaksi (Levent-Levi 2015).

4.4 Acute REST 2.0

Acuten REST-rajapinta on nykyisessä toteutuksessaan versioton, jonka takia rajapinta on vuosien kehityksen myötä alkanut kokemaan kasvukipuja. Rajapinnan julkaistuja kyselyitä ja kyselyiden parametreja joudutaan ajoittain uudistamaan ja vanhoja merkitsemään vanhentuneiksi. Rajapinnan kehitystä saattaa jopa jarruttaa muutoksien mahdollinen yhteensopimattomuus vanhan toteutuksen kanssa. Myös rajapinnan dokumentaatio on nykyisessä toteutuksessaan manuaalisesti ylläpidetty. Rajapinnan laajetessa ja kehitettävien ominaisuuksien kehitystahdin nopeutuessa ohjelmakirjasto alkaa muuttua yhtä haastavammaksi pitää ajan tasalla. Tämän takia on alettu kehittää uutta REST-toteutusta Acuten potilastietojärjestelmään. Uudella toteutuksella tuodaan mahdollisuus rajapinnan versiointiin, joka mahdollistaa helpomman ja joustavamman kehitysmallin. Uudistamisen tarkoituksena on myös toteuttaa helpompi dokumentaation ylläpito, jonka avuksi on kehitysprojektissa valittu vapaata lähdekoodia oleva OpenAPI-spesifikatiota noudattava Swagger-sovelluskehys. Swagger-ekosysteemiin kuuluva toiminnallisuus mahdollistaa lähdekoodista automaattisesti generoituvan dokumentaation. Generoituvan dokumentaation avulla päästään eroon manuaalisesta dokumentaation ylläpidosta.

4.5 OpenAPI-spesifikaatio

OpenApi spesifikaatio (OAS) on määritelmä standardista, joka on ohjelmointikielestä riippumaton kuvauskieli REST-rajapinnoille. Spesifikaatio mahdollistaa REST-rajapinnan täysvaltaisen kuvauksen ilman, että käyttäjällä tai tietokoneohjelmalla on pääsyä itse rajapinnan lähdekoodiin tai erilliseen dokumentaatioon. Tähän kuvaukseen kuuluvat esimerkiksi rajapinnan kutsujen URIt ja kutsuille määritellyt parametrit ja attribuutit, mukaan lukien attribuuttien tyypit ja pakollisuudet. Spesifikaation oleellinen tarkoitus on olla ymmärrettävissä ja käytettävissä sekä ihmisen että koneen näkökulmasta. OpenAPI-kuvauskieli on saavuttanut maailmalla suuren suosion ja on pyrkimyksiensä mukaisesti saavuttamassa kuvauskielten de facto-tittelin. Spesifikaatiota ylläpitävä ja kehittävän organisaation The OpenAPI Initiativen pyrkimys on luoda, kehittää ja julistaa palveluntarjoajasta riippumatonta, neutraalia ja avointa kuvauskieli-standardia. Organisaation perustajajäseniin kuuluvat Google, IBM, ja Microsoft. Tämän jälkeen suuret yritykset kuten Paypal ja Ebay ovat liittyneet jäseniksi (OpenAPI Initiative 2015).

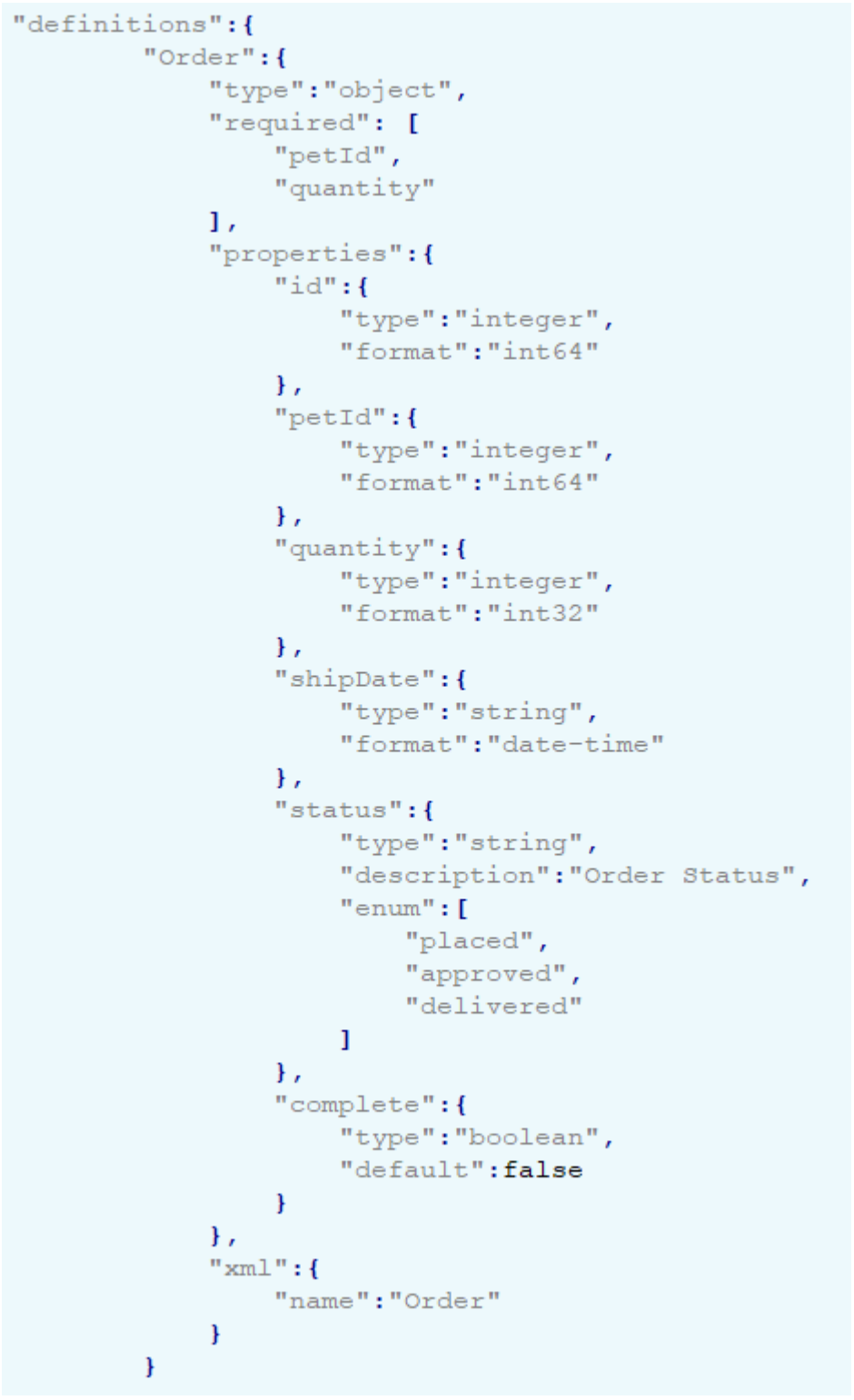
Esimerkkinä kuvauskielestä käytetään usein OpenAPI:n harjoitusmateriaalissa löytyvää petstore-esimerkkiä (kuva 5). HTTP-kutsussa palautunut JSON-tiedosto sisältää lemmikin tilausta kuvaavan objektin.


```
{
"id": 153,
"petId": 38,
"quantity": 2,
"shipDate": "2017-05-11T05:12:30+08:00",
"status": "placed",
"Complete": true
}
```


KUVA 5. Esimerkki tilauksesta JSON-objektina

OpenApi-spesifikaation mukainen esitystapa tästä objektista näkyy seuraavassa kuvassa, josta nähdään, miten jokainen objektin sisältämä atribuutti on oma objektinsa (kuva 6). Atribuutti-objekti sisältää atribuutin tyypin ja vapaaehtoisen format-atribuutin, joka tarkentaa tyyppiä ennestään. Kuvauskielessä käytetyt tyypit ovat määritelty OpenAPI spesifikaatiossa, ja pohjautuvat viralliseen JSON skeemaan. Kuvasta on myös hyvä huomioida, miten objektille pakolliset atribuutit on esitetty required nimisen taulukon sisällä.

Näiden atribuuttien tulee kulkea aina objektin mukana. Kuvauskieli tukee myös oletusarvojen esittämistä, edellyttäen kuitenkin sitä, ettei kyseinen atribuutti ole pakollinen. Viimeisenä merkittävänä huomiona kuvassa voi nähdä miten kuvauskieli sisältää myös tuen luetteloinnille (enumeration), jolla määritellään kaikki atribuutin sallitut arvot.

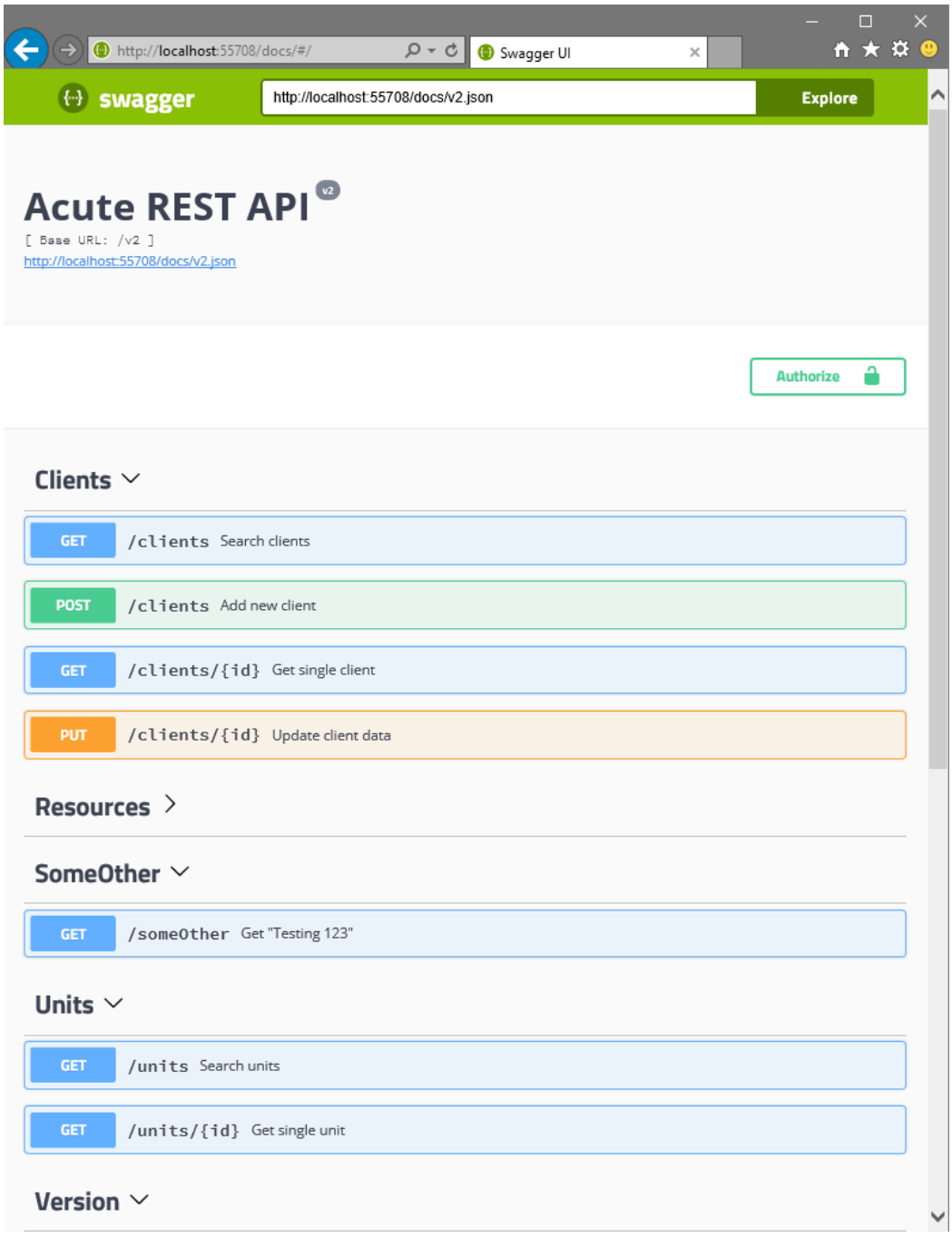


```
"definitions":{
        "Order":{
            "type":"object",
            "required": [
                "petId",
                "quantity"
            ],
            "properties":{
                "id":{
                    "type":"integer",
                    "format":"int64"
                },
                "petId":{
                    "type":"integer",
                    "format":"int64"
                },
                "quantity":{
                    "type":"integer",
                    "format":"int32"
                },
                "shipDate":{
                    "type":"string",
                    "format":"date-time"
                },
                "status":{
                    "type":"string",
                    "description":"Order Status",
                    "enum":[
                        "placed",
                        "approved",
                        "delivered"
                    ]
                },
                "complete":{
                    "type":"boolean",
                    "default":false
                }
            },
            "xml":{
                "name":"Order"
            }
        }
```

KUVA 6. Tilaus-objekti OpenAPI-spesifikaation mukaisesti esitettynä.

OpenAPI-spesifikaation mukaisen kuvauskielen ympärille on alusta asti kehitetty kattavaa “tuoteperhettä”, johon kuuluvat ohjelmistokomponentit hyödyntävät kuvauskieltä erityyppisten ominaisuuksien tuottamiseen. Näihin ominaisuuksiin lukeutuu esimerkiksi

aikaisemmin mainittu ohjelmistokehys SwaggerUI. SwaggerUI generoi OpenAPI-kuvauskielen perusteella dokumentaation sekä verkkosivun REST-rajapinnasta. SwaggerUI esittää tämän interaktiivisen verkkosivun muodossa jossa käyttäjä voi tutkia rajapintaa ja testata eri kutsuja (kuva 7).



KUVA 7. SwaggerUI:n muodostama interaktiivinen käyttöliittymä

Projektin puitteissa oleellisimpiin komponentteihin lukeutuu Swagger Codegen. Swagger Codegen on ohjelmistomoottori, joka mahdollistaa OpenAPI-kuvauskieltä käsittelemällä

API-clientin generoimisen. API-clienttia voi tämän jälkeen käyttää ohjelmakirjaston tapaan rajapintaa kutsuvassa sovelluksessa. Generoitu kirjasto sisältää rajapinnan käyttöön tarvittavat luokat ja metodit. API-clientin generointi mahdollistaa kehittäjille merkittävän resurssien säästön, sillä kirjaston manuaalinen kehitys ja etenkin ylläpito on rajapinnan muuttuessa vaivalloista ja aikaa vievää. Manuaalisen kehityksen työkuorma kasvaa merkittävästi, mikäli vaatimuksena on tarjota kirjasto useammalle ohjelmointikielelle. Swagger-codegen sen sijaan tukee tällä hetkellä API-clienttien generoimista yli 30 eri kielelle kuten C#, Python, PHP, Java, Swift ja TypeScript.

Koska OpenAPI-spesifikaation käyttö oli otettu jo osaksi Acuten seuraavaa iteraatiota, oli tutkimuksen kannalta luonnollista, että tämän teknologian ympärille kehittynyttä ekosysteemiä pyrittiin hyödyntämään myös ohjelmistokirjaston toteutuksessa. Koska käytössä oleva dokumentaatiotyökalu ja Swagger Codegen käyttävät generointiin aina samaa rajapintaa kuvaavaa OpenAPI-spesifikaatiotiedostoa, vastaavat nämä aina toisiaan. Tämä tarkoittaa, että kehittäjän ei tarvitse päivittää dokumentaatiota ja kirjastoa erikseen.

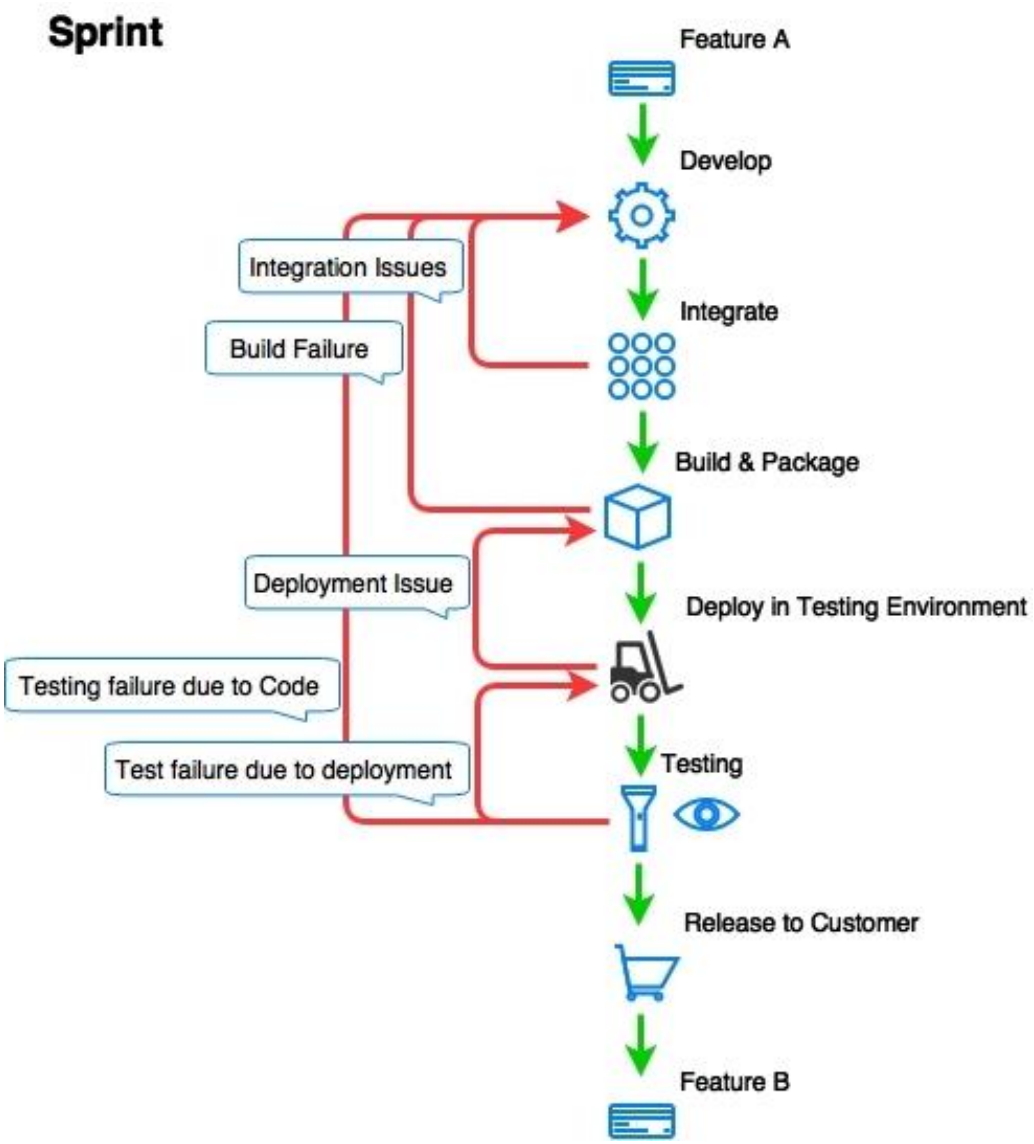
5 TOTEUTUSMENETELMÄT

5.1 Jatkuva Integraatio

Acuten tuotekehityksessä on otettu käyttöön jatkuvan integraation toimintamalli (Continuous Integration). Tämä tarkoittaa kehityksen aikana muodostuvien toiminnallisuuksien välitöntä lisäystä ohjelmiston kehitysversioon. Tämän ansiosta saadaan nopeasti kiinni kehitettävien komponenttien yhteensopivuusongelmat sekä muut riskejä synnyttävät tilanteet. Jotta API clientin laadunvarmistusta voidaan edistää ja kehitysaikoja nopeuttaa, tulee ohjelmakirjaston kehityksen noudattaa tätä toimintamallia. Jatkuvan integraation malli on yksi osa ketterän kehityksen ajatusmallia, jossa jaksottamalla kehitystyö pieniin kokonaisuuksiin nopeutetaan valmiin työn julkaisua ja vähennetään reaktioaikoja ohjelmistokehityksen eri vaiheissa syntyviin ongelmatilanteisiin.

Jatkuvan integraation mallissa pyritään pääsemään eroon ohjelmistoprojekteissa yleisesti ilmaantuvasta ilmiöstä, jossa kehitettävä ohjelma sattaa kokonaisuutena olla pitkiäkin aikoja käyttökelvottomassa kunnossa johtuen hyväksymistestauksen ja integrointivaiheen ajoituksesta vasta tuotteen loppuvaiheeseen. Ongelmia syntyy, kun kehityksen aikana kehittäjät tuottavat omia kokonaisuuksia tuotteelle samalla testaten paikallisesti tuotoksensa omissa haaroissaan (branch). Vasta kehitysvaiheen loppupuolella nämä eri kehittäjien muutokset yhdistetään tuotteeseen ja tuotteen testaus kokonaisuutena alkaa. Tämä aiheuttaa merkittävän määrän lisätyötä integraatioiden loppuunviemiseksi ja lopputestaukselle, jonka aikataulutusta on mahdotonta suunnitella etukäteen. (Humble & Harley 2011, 55).

Jatkuvassa integraatiossa tilanne on toinen. Jatkuvan integraation mallissa integraatioaika siirtyy kehityksen venyvistä loppuhetkistä jatkuvaksi, minuutteja vieväksi prosessiksi kehityksen aikana. Heti kun kehittäjä tekee ohjelmaan muutoksen, käännetään ja ajetaan koko ohjelma siihen kirjoitettujen automaatiotestien kera. Tämä johtaa tilanteeseen jossa koko kehitystiimi saa heti tietoonsa ohjelmistoon mahdollisesti ilmaantuvat virheet ja yhteensopivuusongelmat. Siinä missä tavanomaisessa ohjelmistokehityksen vesiputousmallissa ohjelmisto saataa olla virheellinen pitkiäkin aikoja, on jatkuvan integraation mallissa tavoitteena pitää kehitettävä ohjelma käyttökunnossa läpi kehityksen (Humble & Harley 2011, 55). Ennen ohjelmoidun muutoksen päätymistä hyväksymistestaukseen muutos käy monen eri vaiheen läpi, joissa mahdolliset ongelmatilanteet saadaan kiinni ja niihin voidaan reagoida välittömästi (kuva 8).



KUVA 8. Jatkuvan integraation malli ketterässä kehitysmallissa. (Pathania 2016).

Jatkuvan integraation saavuttamiseksi on kuitenkin huomioitava mallin laatua ja tehoa parantava vaatimus. Vaikka jatkuvan integraation mallissa ohjelmaan tulleiden muutosten aiheuttamat käännösvirheet saadaankin kiinni, ei mallin laadunvarmistus ole riittävällä tasolla pelkästään tällä toiminnallisuudella. Mikäli automaatiotestien kirjoitusta ei ole otettu käyttöön mukaan työprosessiin, ei malli yllä varmentamaan ohjelmiston toimintoja regressiivisesti. Vaikka ohjelma kääntyykin ohjelman havaitessa muutoksen, tulisi sen suorittaa aina automaatiotestit regressiovirheiden havaitsemiseksi. Tällä tavoin mallia noudattavat tiimit löytävät virhetilanteet nopeammin mikä vähentää korjausaikoja, projektien aikataulujen venymistä sekä resurssien rasitusta.

5.2 Teamcity

Jatkuvan integraation mallia suorittamaan on Acutessa otettu käyttöön JetBrainsin kehittämä Teamcity-palvelinohjelmisto. Teamcity julkaistiin vuonna 2006 ja se pohjautuu Java-ohjelmointikieleen. Teamcityssä skripteillä toteutettavat konfiguroinnit mahdollistavat laajan kirjon erilaisia toimintoja kuten asennettavan ohjelmiston kääntöprosessien teon erilaisten laukaisimien (trigger) avulla, mitkä määrittävät milloin mikäkin toiminto ajetaan. Esimerkkinä tälläisestä laukaisijasta on uuden muutoksen havaitseminen versionhallinnassa, mikä laukaisee koko ohjelman kääntämisprosessin. Teamcity toimi myös tämän projektin keskeisenä työkaluna, missä API clientin generoinnin automaatio pystyttiin ohjelmoimaan omaksi osaksi integraatioprosessia. Generoinnin lisäksi API clientin julkaisu ohjelmakirjastojen jakamiseen tarkoitetulle palvelimelle ja julkiseen Git-repositoryyn toteutettiin Teamcityn sisällä käyttäen Powershell-skriptejä.

5.3 Windows Powershell

Projektin vaatimat automaatioskriptit tuotettiin pääsääntöisesti PowerShellin avulla. Windows Powershell on Microsoftin kehittämä, vuonna 2006 julkaisema .Net Frameworkiin perustuva komentorivityökalu, joka käsittää oman komentotulkin (shell) sekä komentosarjakielen (scripting language). Powershell kehitettiin korvaamaan Windowsin vanhan komentotulkki cmd.exen komentorivityökaluna. Powershell sisältääkin cmd.exen kaikki toiminnot, mutta on toiminnallisuudeltaan merkittävästi laajempi. Powershellin käyttötarkoitusta on kehitetty toimimaan työkaluna toistuvien ja raskaiden ylläpitotehtävien selvittämisessä ja automatisoinnissa. Powershellin muista komentotulkeista erottavia ominaisuuksia ovat muun muassa C#-kielen kaltainen syntaksi, .NET Framework luokkien hyödyntäminen cmdlet-komentojen kautta sekä pääsy Windowsin omiin tietolähteisiin kuten järjestelmän tietokantaan ja tiedostorakenteeseen (Ford 2007. 7). Elokuussa 2016 Microsoft julisti tehneensä Powershellin avoimen lähdekoodin ohjelmaksi, sisältäen tuen monialustaisuudelle (cross-platform). Powershell Coreksi nimetty avoimen lähdekoodin versio on saatavissa Github-repositoryssa.

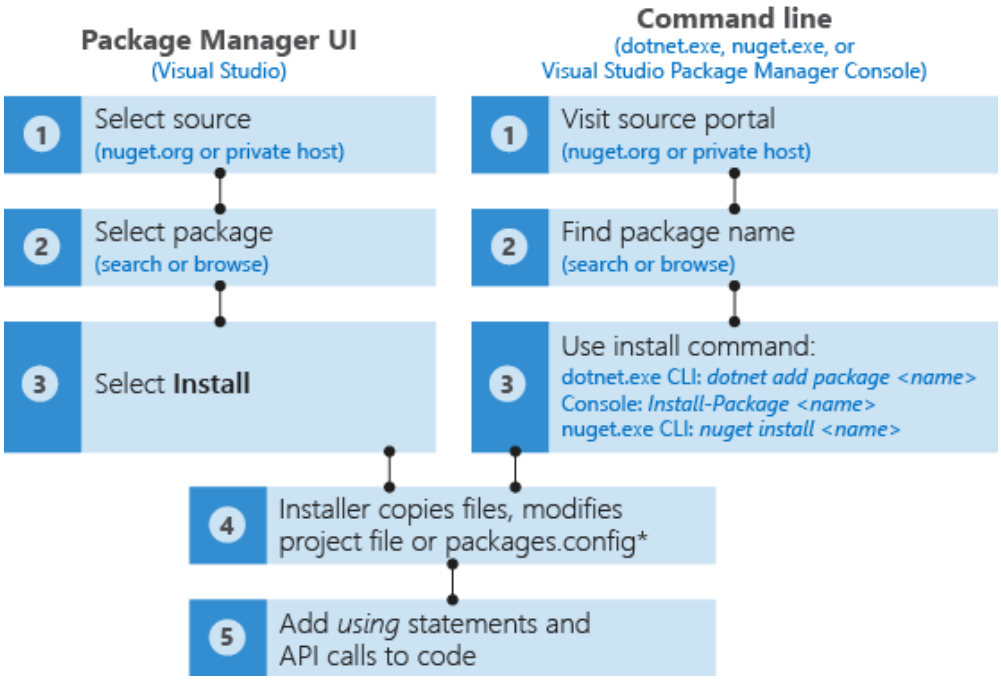
5.4 NuGet

Projektin työkaluksi valittiin Swagger Codegen, jonka ominaisuuksiin kuuluu kirjastojen generointi kymmenille eri ohjelmointikielille. Tästä huolimatta kirjasto julkaistaan projektinaikaista määrittelyä mukaillen vain C#-kielellä. C#-kirjaston julkaiseminen käännetyssä muodossa toteutettiin käyttämällä Visual Studion sisältämää NuGet-paketointiohjelmaa. NuGet on Microsoftin kehittämä .NET-frameworkin kanssa toimiva paketinhallintaohjelma. Sen pääasiallinen käyttötarkoitus on toimia työkaluna ohjelmapakettien luonnissa, haussa ja käytössä.

Yleiskäyttöisen koodin kehittäminen, jakaminen ja käyttö ovat olennainen osa moderneja kehitysalustoja. On tyypillistä, että tälläiset koodikokonaisuudet koostetaan omiin paketeihin, jotka sisältävät käännetyn koodin. NuGet on Microsoftin kehittämä mekanismi, joka määrittelee tavan näiden pakettien luomiseen, jakamiseen sekä käyttöön .NET frameworkillä (Brockschmidt & Myers 2018).

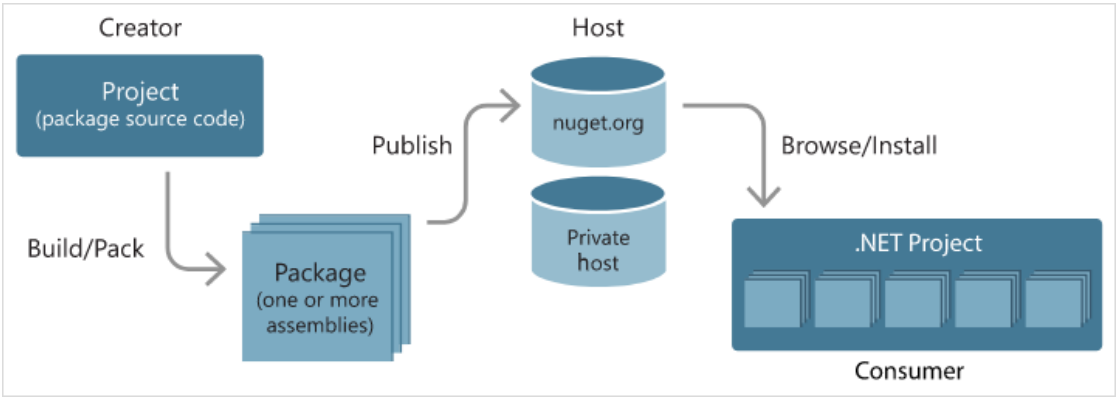
Nuget-pakettien jakaminen tapahtuu niille varatuilla NuGet-palvelimilla, jotka voivat olla joko julkisia tai yksityisiä. Pakettien julkaiseminen yksityisillä palvelimilla mahdollistaa esimerkiksi yrityksille tavan jakaa uudelleen käytettävää lähdekoodiaan eri projektien välillä helpommin. Oli NuGet-palvelin joko yksityinen tai julkinen, noudattaa pakettien käyttöönotto kuvan 9 osoittamaa tapaa. Kuvassa voidaan nähdä Nuget-paketin käyttöönottoprosessi Visual Studion pakettimanageria tai komentoriviä hyödyntäen.

NuGet tarjoaa oman julkisen pakettipalvelimen osoitteessa nuget.org. Tämä palvelin omaa yli 100,000 pakettia joita käyttävät miljoonat kehittäjät ympäri maailmaa (Brockschmidt & Myers 2018). Kuten edellä jo mainittiin, NuGet-pakettipalvelimien ylläpito on kuitenkin mahdollistettu lähes mihin tahansa sijaintiin, kuten sisäiseen verkkoon tai omalle levylle.



KUVA 9. NuGet-paketin käyttöönottoprosessi (Microsoft 2018)

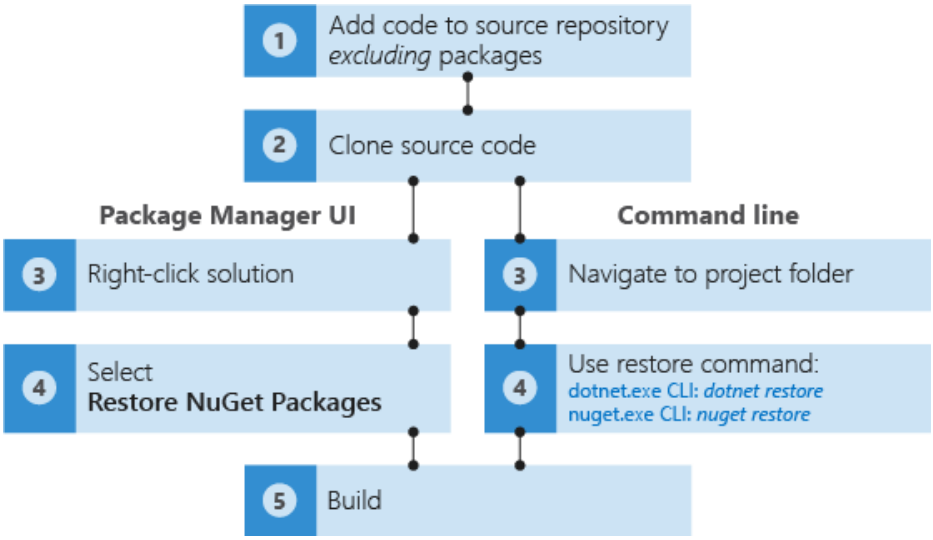
Pakettipalvelimen tehtävä on toimia linkkinä paketin tekijän ja pakettia hyödyntävien kehittäjien välillä. Paketin julkaisuprosessi etenee lähdekoodin paketoinnin jälkeen NuGet-palvelimelle, josta pakettia hyödyntävä kehittäjä etsii ja lataa paketin omaan projektiinsa (kuva 10).



KUVA 10. Paketin matka kehittäjältä sen käyttäjälle. (Microsoft 2018)

Vaikka kehittäjä hyödyntää NuGet-paketteja kehittämässään sovelluksessa, ei näitä paketteja yleensä tallenneta versionhallintaan. Kun toinen kehittäjä hakee projektin omaan kehitysympäristöönsä, tulee tämän ladata tarvittavat paketit itse (Brockschmidt 2018).

Tieto tarvittavista paketeista kuitenkin sisällytetään sovelluksessa joko manuaalisesti tai käyttäen NuGetin generoimaa packages.config konfiguraatiotiedostoa (kuva 11)



KUVA 11. Sovelluksen pakettien uudelleenpalautusprosessi (Microsoft)

6 KÄYTTÖÖNOTTO JA PROSESSIT

6.1 Kehitys

Kirjaston generoituminen voidaan jakaa kolmeen päävaiheeseen.

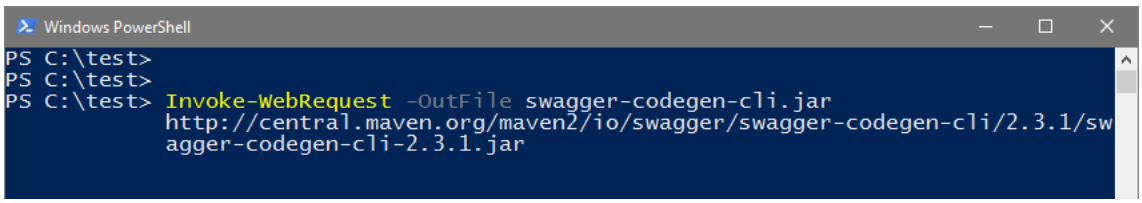
- REST rajapinta-sovelluksen käynnistäminen
- Swashbuckle-framework generoi sovelluksen lähdekoodista OpenAPI- spesifikaatiotiedoston
- Swagger Codegen generoi spesifikaatiotiedostosta ohjelmistokirjaston

6.1.1 Swashbuckle

Swashbuckle on C# ohjelmakirjasto, joka generoi OpenAPI spesifikaation JSON-tiedostoon .NET Web API projekteista. Kun rajapinnan sovellus käynnistetään, Swashbuckle käy läpi rajapinnan lähdekoodin luokat ja metodit, joiden perusteella se generoi JSON-formaattia olevan spesifikaatiotiedoston. Swashbuckle oli otettu käyttöön jo Acuten rajapinnan uuteen iteraatioon, jossa sitä käytettiin SwaggerUI-työkalun generoimiseen. Tämän projektin myötä sen generoimaa OpenApi-spesifikaatiota voitiin hyödyntää myös ohjelmistokirjaston generoimiseen.

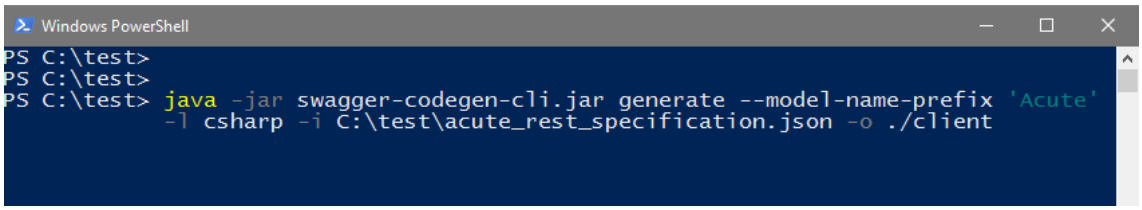
6.1.2 Swagger Codegenin käyttöönotto

Ensimmäisenä askeleena projektin käytännön toteutuksessa tuli asentaa Swagger-codegen työkalu. Kehityksen aikana suurin osa työkalun käytöstä tapahtui paikallisesti työkoneella, jossa tarvittavat komennot voitiin todeta toimiviksi ennen automatisoituun kääntämisprosessiin siirtoa. Swagger-codegenin hankkimiseen verkosta käytettiin PowerShellin cmdlet-pyyntöä Invoke-WebRequest:ia, jolla työkalu haettiin Maven 2 Central Repository-tietokannasta (kuva 12). Invoke-WebRequest tekee HTTP- ja HTTPS-pyyntöjä verkkosivuille ja -palveluihin, parsii vastaussanoman ja palauttaa sivulta palautuneen materiaalin.



KUVA 12. Swagger Codegenin lataus Maven-repositorysta

Kun Swagger Codegen on ladattu haluttuun tiedostohakemistoon, on se valmis generoimaan ohjelmakirjastoja halutuilla kielillä. Generointi tapahtui komentorivillä syöttämällä -i option avulla työkalulle rajapintaa kuvaavan, OpenAPI-spesifikaation sisältävän JSON-tiedoston. Tämän tiedoston generoitumiseen vaikuttavia toimenpiteitä käydään läpi seuraavassa kappaleessa. Komentorivillä generointikomennolle voi antaa useita eri optioita, joilla voidaan vaikuttaa generoituun lopputulokseen. Tyypillisten tulostepolkujen (output path) määrittämisen lisäksi optioihin kuuluu esimerkiksi generoidun kirjaston malliluokkien etu- ja/tai takaliitteiden määrittäminen (kuva 13). Eri ohjelmointikielille spesifiset optiot tulee kuitenkin syöttää komentorivillä optiolla -c, jolle annetaan arvoksi polku JSON-tyyppiseen konfiguraatiotiedostoon.



KUVA 13. API clientin generointikomento optioilla

6.1.3 Kutsun kehityskaari lähdekoodista kirjaston metodiksi

Kun rajapintaan kehitetään uusi kutsu, tuli kehityksen aikana ottaa huomioon miten lähdekoodia muokkaamalla spesifikaatio-tiedosto muuttui halutun malliseksi. Swagger Codegen generoi kutsut oletuksena formaattin, jota ei koettu Acuten tuotteelle sopivaksi. Oletusformaatissa rajapinnan GET /clients/{id} muotoinen pyyntö kääntyi ClientsByIdGet-metodiksi generoituneessa kirjastossa. Asiakkaan päivityksessä metodin nimi oli ClientsByIdPut, kun kutsun muoto oli PUT /clients/{id}. Näistä esimerkeistä voidaan päätellä yleinen kaava miten OpenAPI-spesifikaation operaatioiden nimet generoituvat

Swashbucklea käytettäessä. Operaatioiden nimien muokkaus onnistui kuitenkin lisäämällä kutsuihin SwaggerOperation-attribuutti, jolle pystyi antamaan kuvaavammat nimitykset kuten GetClientsById ja UpdateClient (kuva 14).

```
/// <summary>
/// Update client data
/// </summary>
/// <param name="id">client's id</param>
/// <param name="clientData">client data</param>
[SwaggerOperation("UpdateClient")]
[HttpPut("{id}")]
0 references | 1 author, 1 change
public Client Put(int id, [FromBody]Client clientData)
{
    Client client = Get(id);
    client.UpdateClient(clientData);
    return client;
}
```

KUVA 14. Rajapinnan lähdekoodiin kirjoitettava syntaksi spesifikaation luomiseksi

Kun lähdekoodiin kutsuille merkittiin parametrien nimet, kuvaukset ja operaatioille määritellyt nimitykset, muodostui kuvaustiedostoon sitä vastaava operaatio (kuva 15). Kuvassa voidaan nähdä miten aikaisemman kuvan näköisestä metodista muodostui OpenAPI-spesifikaation mukainen operaatio.

```
"/clients/{id}": {
    "put": {
        "tags": ["Clients"],
        "summary": "Update client data",
        "operationId": "UpdateClient",
        "parameters": [{
            "name": "id",
            "in": "path",
            "description": "client's id",
            "required": true,
            "type": "integer",
            "format": "int32"
        },
        {
            "name": "clientData",
            "in": "body",
            "description": "client data",
            "required": false,
            "schema": {
                "$ref": "#/definitions/Client"
            }
        }],
        "responses": {
            "200": {
                "description": "Success",
                "schema": {
                    "$ref": "#/definitions/Client"
                }
            }
        }
    }
```

KUVA 15. Client-resurssin PUT-operaatio spesifikaatiotiedostossa

Kun spesifikaatiotiedosto oli saatu näyttämään Acuten standardien mukaiselta, generoidun ohjelmistokirjaston käyttö demonstroitiin kutsumalla syntyneitä metodeita testisovelluksessa (kuva 16). Prosessin aikana tuli selväksi, että on tärkeää huomioida kutsujen operaatioiden nimeäminen kehitysvaiheessa aina uusia kutsuja tehdessä. Mikäli tämän vaiheen unohtaa, on näiden operaatioiden jälkeenpäin nimeäminen haastavaa kirjastoa käyttävien osapuolien kannalta.

```
static void Main()
{
    var config = GetConfiguration();

    var clientInstance = new ClientsApi(config);

    try
    {
        var myClient = clientInstance.GetClientById(123);

        myClient.City = "Tampere";

        clientInstance.UpdateClient(123, myClient);
```

KUVA 16. Asiakkaan päivitys generoidun ohjelmakirjaston avulla

6.2 Automatisointi

Työprosessin automatisointi tapahtui TeamCity-palvelinohjelmistolla. Automatisointi suunniteltiin kahteen vaiheeseen. Ensimmäisenä vaiheena on ohjelmakirjaston generointi, jossa kirjasto luodaan rajapinnasta syntyneen kuvauskieli-tiedoston perusteella. Toinen vaihe on julkaisuprosessi. Julkaisuprosessiin kuului NuGet-paketin luonti, jossa generoitunut kirjasto paketoidaan NuGet-palvelimelle syötettäväksi paketiksi. Toinen vaihe on generoidun lähdekoodin tallentaminen Git-repositoryyn, josta kirjaston käyttäjät voivat ladata muokattavan lähdekoodin omaan kehitysympäristöön.

Teamcity-ohjelmistoa käyttäen ohjelmakirjaston automatisointi osoittautui helpoksi. Toimeksiantajan Teamcity-ympäristössä automatisoidut prosessit ovat jaettu omiin projekteihinsa. Projektit koostuvat prosessivaiheista (build step), joiden sisältämiä komentoja syötetään käännöspalvelimien “agenteille” (build agent). Teamcityä voikin pitää eräänlaisena tehtävien jakajana, joka käskyttää käännöspalvelimia tekemään haluttuja toimintoja määritellyssä järjestyksessä.

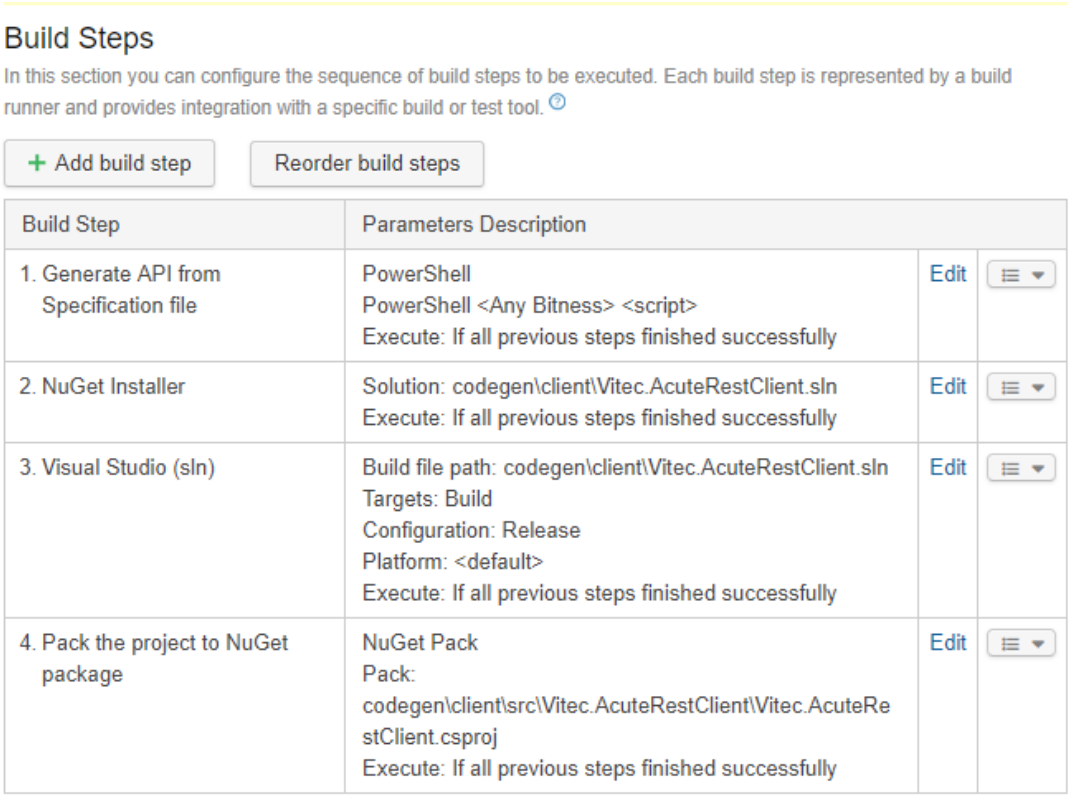
6.2.1 Generoinnin automatisointi

Kun ohjelmakirjaston automaattista generoimista suunniteltiin, tuli julkaisussa kiinnittää erityistä huomiota prosessin optimointiin. On selvää, että työkalusta tulee julkaista uusi

versio aina kun rajapinta kokee muutoksia. Kysymys kuitenkin syntyi siitä, onko kirjasto tarpeellista generoida ja julkaista uudelleen, jos vain rajapinnan lähdekoodi muuttuu. Teknisesti ajatellen generointia ei tarvitse tehdä, sillä generaattori välittää vain rajapinnan kuvaavasta JSON-tiedostosta. Rajapinnan kuvaava JSON-tiedosto muuttuu vain rajapinnan muuttuessa. Jos kirjasto halutaan generoida kun rajapinta muuttuu, luo se tarpeen rakentaa uusi toiminnallisuus automaatioprosessiin. Toiminnallisuuden tulisi vertailla lähdekoodista syntynyttä kuvauskielitiedostoa edelliseen versioon samasta tiedostosta aina lähdekoodin muuttuessa. Muita rajapinnan kehittäjiä konsultoimalla päädyttiin toiminnallisuuden tarpeen sivuuttamiseen. Kun generoinnin suorituskykyä tarkasteltiin, nähtiin kirjaston generoinnin olevan suhteellisen vähän resursseja vaativa operaatio. Koska generointi ei kuormita automatisointiprosessia merkittävästi, tiedostoja vertailevan toiminnallisuuden ajateltiin tuovan enemmän monimutkaisuutta kuin hyötyä prosessiin. TeamCityn automatisointiprosessi suunniteltiin julkaisemaan generoitu työkalu aina kun uusi ohjelmistopaketti Acuten rajapinnasta julkaistaan. Koska OpenAPI-spesifikaation käyttöön ottavan REST-rajapinnan uusi iteraatio oli projektin ajan vielä prototyyppivaiheessa, ei yöllisessä käännösprosessissa generoitunut automaattisesti työkalun tarvitsemaa JSON-tiedostoa. Jotta Swagger Codegen saatiin konfiguroitua mahdollisimman todenmukaista kehitystilannetta muistuttavaan käännösprosessiin, generoitiin spesifikaation JSON-tiedosto ensin paikallisesti. Tämän jälkeen se sijoitettiin yhdelle kehityspalvelimista, josta tiedoston sijainti osoitettiin generaattorille PowerShell-komentosarjassa generoinnin yhteydessä.

6.2.2 Julkaisun automatisointi

Ohjelmistokirjaston julkaisun suunnittelu osoittautui helpoksi. Teamcity sisältää tuen oman NuGet-palvelimen ylläpidolle, joka mahdollisti nugettipaketin paketointi- ja julkaisuprosessin merkittävän suoraviivaistamisen. Ohjelmistokirjaston Teamcity-projektiin lisättiin uusi prosessivaihe, johon konfiguroitiin polut ohjelmistokirjaston sijaintiin (kuva 17).



KUVA 17. Teamcityn prosessivaiheiden rakennus

Tämän lisäksi valittiin prosessin tuloksena muodostuvien artifaktien julkaisu, mikä tarkoitti NuGet-paketin julkaisua Teamcityn sisäiselle NuGet-palvelimelle. Julkaisun jälkeen kirjaston oli mahdollista ladata kirjasto palvelimelta omaan projektiin käyttämällä esimerkiksi Visual Studion nuget-manageria. Jotta kirjaston toimittaminen käyttäjille olisi mahdollisimman monimuotoista, suunniteltiin ohjelmakirjaston lähdekoodi julkaistavaksi julkiseen versionhallintaan. Kun ohjelmakirjasto on tuotantovalmis, luodaan Teamcityyn uusi prosessivaihe versionhallintaan julkaisua varten. Prosessivaihe toteutetaan komentorivi-komennoilla, joilla haetaan generoidun ohjelmistokirjaston lähdekoodi ja tallennetaan tämä merkattuna (tagging) versionhallintaan.

7 YHTEENVETO

Kun arvioidaan opinnäytetyön onnistumista kokonaisuutena, on tärkeää tarkastella työn alussa asetettujen tavoitteiden kohtaamista työn tuotosten kanssa. Perimmäisenä tarkoituksena oli löytää keino REST-rajapinnan käyttöön ohjelmakirjaston kautta. Lisävaatimuksena tälle tehtävälle oli saavuttaa kirjaston helppo ylläpito ja julkaisu ulkopuolisille kehittäjille. Työn tuloksena saavutettiin prosessi, jossa rajapinnan lähdekoodin perusteella generoidaan ohjelmakirjasto. Prosessiin sisältyy generoinnin lisäksi automaattinen julkaisuvaihe, joka julkaisee kirjaston julkiseen käyttöön. Kun vaatimuksia ja konkreettisia tuloksia vertailee keskenään, voidaan todeta tavoitteiden täyttyneen. Työn aikana tutkittiin kuinka API clientin tarjoamisen haittapuolet, kuten manuaalisen ylläpidon vaatima työmäärä selätettäisiin. Projektin alussa osoitettiin huoli REST-rajapinnan muutoksista aiheutuvasta lisätyöstä, mikäli tuotettava ohjelmakirjasto ei hallitse näitä muutoksia automaattisesti. Opinnäytetyön tuotoksesta ei haluttu uutta jatkuvaa ylläpitoa vaativaa ohjelmistokokonaisuutta. Tähän löydettiin ratkaisu tutustumalla rajapinnan lähdekoodista automaattisesti kirjastoja generoiviin työkaluihin. Työkalut mahdollistivat ohjelmakirjaston generoimisen samalla työprosessilla, jolla rajapinnan dokumentaatio ylläpidetään. Ohjelmakirjaston muodostumista automaattisesti lähdekoodista voidaankin pitää tämän opinnäytetyön suurimpana onnistumisena. Kun projekti aloitettiin, ei ohjelmakirjaston julkaisuprosessista ollut ennakkomäärittelyä. Tavoitteena oli saada kirjasto julkaistua vapaana lähdekoodina, mutta rajapinnan lähdekoodia ei haluttu paljastaa. Tämä tavoite saavutettiin käyttöönotettujen generoimistyökalujen avulla. Generoidut tuotokset muodostuvat rajapintaa kuvaavasta tiedostosta, joka sisältää vain rajapinnan dokumentaation omassa formaatissaan. Käyttöönotettu generoimistyökalu on yksinkertainen ratkaisu monimutkaiseen tehtävään. Ilman automaattista generointia samaan lopputulokseen pääsy olisi ollut erittäin laaja projekti vaatien mittavan määrän resursseja.

Projektia työstettäessä opittiin useita asioita. Yksi tärkeimmistä opituista asioista oli se, että kaikkea ei tarvitse tehdä itse. Kun alustava selvitys projektin työmenetelmistä alkoi, ei ennakkotietoa automaattisesta ohjelmistokirjaston generoinnista ollut toimeksiantajalla eikä kehittäjällä. Alustavassa suunnittelussa ohjelmistokirjaston luonti uskottiin olevan tarpeellista toteuttaa itse. Kun selvitys käytettävistä menetelmistä alkoi, huomio kuitenkin kiinnittyi juuri rajapintoja varten kehitettyihin automaattisiin työkaluihin. Työkalujen an-

siosta projektissa pystyttiin allokoimaan ajankäyttöä enemmän automatisoinnin suunnitteluun ja generoituneen ohjelmakirjaston viilaukseen Acute-tuoteperheeseen sopivaksi. Projektin edetessä huomattiin, että jatkuvan integraatio -mallin toteutumiseksi tulee kirjaston luonti sisällyttää Acuten rajapinnan kehitysprosessin sisälle. Kirjaston generoinnin ollessa riippuvainen rajapinnasta, ei näitä kahta tuotetta voi eriyttää toisistaan. Automatisointiin pitikin suunnitella generoinnin laukaisu aina kun rajapinta käännetään. Projektin kannalta parantamisen varaakin löytyi. Koska Acuten tuotekehitykseen yritetään sisällyttää laajenevassa määrin automaatiotestausta, olisi sitä pitänyt painottaa enemmän projektin suunnittelussa. Jos projektissa olisi voitu jotain tehdä toisin, olisi se ollut testisovelluksen kirjoittaminen. Kirjoittamalla ohjelmakirjastolle yksikkötestejä projektin alusta lähtien olisi automaatiotestauksen yleinen käyttöönotto saanut tuulta alleen ja näin tuonut kaivattua varmuutta kirjaston laadusta. Toinen asia mitä projektissa olisi voitu tehdä toisin oli Acuten rajapinnan uuden version kehitys ennen muuta käytännön toteutusta. Jotta julkaisuprosessi olisi voitu tehdä täysin tuotantovaihetta vastaavaksi kokonaisuudeksi, olisi rajapinnan uusi iteraatio pitänyt kehittää pidemmälle. Projektin alussa olisi voitu toteuttaa ne pääkohdat rajapinnasta kuten sisäiseen versionhallintaan sijoittaminen sekä yöllinen käännösprosessi. Tämän jälkeen kirjaston generointi olisi voitu sitoa rajapinnan käännösprosessiin tiukemmin, ilman että generoinnin vaatimia tiedostoja olisi tarvinnut manuaalisesti sijoittaa käännöspalvelimille. Mikäli tämä olisi toteutettu oikein, jatkuvan integraation mallia olisi päästy toteuttamaan alusta lähtien. Projektia tehdessä opittiin myös aikataulun suunnittelua. Kun käytettävät työkalut olivat päätetty, työmäärän arviointi käytännön toteutuksesta alkoi. Tässä vaiheessa projektia olisi ollut aiheellista jakaa työ pienempiin kokonaisuuksiin, jolloin työmäärän arviointi olisi ollut helpompaa. Työn onnistumista voidaan mitata myös ammatillisen osaamisen kasvatuksella. Projektin aikana kootun aineiston ansiosta asiantuntemus rajapintojen kanssa työskentelystä karttui mittavasti. Myös jatkuvan integraation malliin tutustuminen tuotti tärkeää pohjatietoa kehitysprosesseista ohjelmistotuotannossa.

Työn tuloksista voi vetää myös selviä johtopäätöksiä. Koska rajapintoja kehitetään jatkuvasti uusiin tarkoituksiin, niiden hyödyntämiseen kehitetään yhä uudenlaisia ratkaisuja. Rajapinnat muodostavat modernissa ohjelmistokehityksessä linkin sovelluksien välille. Rajapintojen ympärille luodut teknologiat ovat tarkoitettu helpottamaan rajapintojen kanssa työskentelyä tarjoamalla automatisoituja prosesseja poistaen manuaalista työtä. Opinnäytetyön tuloksista voikin vetää johtopäätöksen vahvasta todennäköisyydestä, että

rajapintojen käyttön helpottuminen kehittyy entisestään tulevaisuudessa. Avoimen lähdekoodin ja toimittajavapaat teknologiat kuten OpenAPI-kuvauskieli kiihdyttävät verkkopalveluiden kehittymistä maailmalla, mahdollistaen yhä uusien käyttötapausten syntymisen. Työn aikana vahvistui myös käsitys siitä, kuinka tärkeää jatkuvan integraation sisällyttäminen ohjelmistokehitykseen on. Kirjaston ollessa rajapinnasta syntyvä “sivutuote”, kirjaston erillinen testaus jäisi hyvin suurella todennäköisyydellä huomioimatta kehittäjän tehdessä rajapintaan muutoksia. On selvää, että jatkuvan integraation mahdollistama automaattinen laadunvarmistus vie suuren osan yhteensopivuushuolista pois julkaisuvaiheessa.

Vaikka työ saavuitti sille määritellyt tavoitteet, jatkokehitykselleKkin jäi varaa. Kun Acuten rajapinnan seuraava iteraatio siirtyy tuotantoon, tulee ohjelmakirjaston automaatioprosessia muokata. Työn aikana käytettiin käännöspalvelimelle sijoitettua staattista spesifikaatiotiedostoa automaatioprosessissa. Tuotantovaiheessa tämä tiedosto kuitenkin generoituu rajapintasovelluksen käynnistyksen yhteydessä. Jotta kirjasto pysyy mukana rajapinnan muutoksissa, on kirjaston generoinnin hyödynnettävä rajapintasovelluksen generoimaa spesifikaatiotiedostoa. Työtä on myös mahdollista jatkaa pidemmälle kirjoittamalla automaatiotestejä kirjaston toiminnoille. Kirjoittamalla automaatiotestejä vähennetään huolta muutoksien tuomista virheistä ja päästään lähemmäksi jatkuvan integraation mallia. Jos kirjastoon ilmaantuu logiikan rikkova virhe, automaatiotestaus antaa tästä välittömästi ilmoituksen. Koska ohjelmakirjaston lähdekoodi on tarkoitus julkaista julkiseen versionhallintaan, avaa se mahdollisuuden tarjota kirjaston lähdekoodi myös muilla ohjelmointikielillä. Koska Acuten tuotekehitys on kohdistautunut käyttämään palvelinpuolen koodissa pääsääntöisesti C#-ohjelmointikieltä, ei resursseja riitä yhden työkalun tukemiseen muilla ohjlemointikielillä. Toisaalta ohjelmistokirjaston tarjoaminen usealla kielellä vaikuttaa merkittävästi kirjaston kohdeyleisön laajenemiseen. Jatkokehityksessä onkin mahdollista määritellä esimerkiksi C#-kielen kirjasto teknisen tuen piiriin, mutta kirjasto voidaan tarjota ilman tukea myös muilla suosituilla kielillä kuten Java ja Swift. Kun ajatellaan muita erillisiä jatkotoimia kirjastolle ja sen julkaisulle, on syytä suunnata katse myös käyttäjien suuntaan. Koska kirjasto tehtiin nimenomaisesti ulkopuolisten kehittäjien käyttöön, on syytä hankkia palautetta myös käyttökokemuksista. Tieto onko kirjaston käyttö helpoittanut rajapinnan kanssa työskentelyä ja miten kirjaston käyttökokemusta voidaan parantaa, ovat pääasiallisia kysymyksiä joiden vastauksista toimeksiantaja

hyötyy. Kun projektin jatkotoimissa otetaan huomioon käyttäjien antama palaute, voidaan kirjastoa mukauttaa täyttämään toiveet ja saavuttaa kirjaston täysi potentiaali kehittäjien työkaluna.

8 LÄHTEET

Amundsen, M.,Richardson, L. 2013. RESTful Web APIs. Sebastopol: O'Reilly Media, Inc

Alfame n.d. Avain ketterän liiketoimintastrategian toteuttamiseen - API/Ohjelmointirajapintaopas IT-päättäjille. Luettu 18.04.2018. Lataus vaatii yhteystiedot. https://www.alfame.com/avain-ketteran-liiketoimintastrategian-toteuttamiseen-ohjelmointirajapintaopas

Brockschmidt, K., Myers, A. 2018. An introduction to NuGet. Microsoft. Luettu 18.4.2018 https://docs.microsoft.com/en-us/nuget/what-is-nuget

Brockschmidt, K. 2018. Package consumption workflow. Microsoft. Luettu 18.4.2018 https://docs.microsoft.com/en-us/nuget/what-is-nuget

Fielding, R.T. 2000. Architectural Styles and the Design of Network-based Software. University of California. Väitöskirja

Ford, J.L. 2007. Microsoft Windows Powershell Programmming for the absolute beginner. Boston: Thomson Course Technology PTR.

Hautamäki, V. Acusfääri - Rajat auki kehityksen nimissä. Luettu 18.4.2018 http://www.acute.fi/blog/2017/02/acusfaari-rajat-auki-kehityksen-nimissa/

Humble, J., Harley D. 2010. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley Professional

Koho, S. 2017. Etävastaanotot yleistyvät. Potilasturvallisuutta ei vaaranneta. Forum24. Luettu 2.4.2017. https://www.forum24.fi/sivut/artikkeli/18697/Etvastaanotot-yleistyvt_-Potilasturvallisuutta-ei-vaaranneta

Levent-Levi, T. 2015. Why an SDK is Critical to your API Offering. Luettu 20.4.2018. https://bloggeek.me/sdk-critical-to-api/

Mason, R. 2011. How REST replaced SOAP on the Web: What it means to you. Luettu 15.4.2018. https://www.infoq.com/articles/rest-soap

Massé, M. 2011. REST API Design Rulebook. Sebastopol: O'Reilly Media, Inc

OpenAPI Initiative 2015. New Collaborative Project to Extend Swagger Specification for Building Connected Applications and Services. Luettu 15.4.2018 https://www.openapis.org/announcement/2015/11/05/new-collaborative-project-to-extend-swagger-specification-for-building-connected-applications-and-services

Oracle 2013. The Java EE 6 Tutorial - What Are RESTful Web Services? Luettu 18.4.2018 https://docs.oracle.com/javaee/6/tutorial/doc/gijqy.html

Pathania, N. 2016. Learning Continuous Integration with Jenkins. Packt Publishing

Richardson, L., Ruby, S. 2007. RESTful Web Services. Sebastopol: O'Reilly Media, Inc

Sarrel, M. 2016. Why API Providers Should Offer SDKs. Luettu 20.4.2018 https://www.programmableweb.com/news/why-api-providers-should-offer-sdks/analysis/2016/02/24